

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE				
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				
AD-A217 923			5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA-89-006				
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT Univ of Texas - Austin		6b. OFFICE SYMBOL (If applicable)		7a. NAME OF MONITORING ORGANIZATION AFIT/CIA			
6c. ADDRESS (City, State, and ZIP Code)				7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6583			
8a. NAME OF FUNDING / SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8c. ADDRESS (City, State, and ZIP Code)				10. SOURCE OF FUNDING NUMBERS			
				PROGRAM ELEMENT NO.		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) (UNCLASSIFIED) A Library of Modular Routines for Generating Test Patterns for Digital Circuits							
12. PERSONAL AUTHOR(S) Thomas Humbert Belvin, Jr							
13a. TYPE OF REPORT THESIS/DOCUMENT		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1988		15. PAGE COUNT 57	
16. SUPPLEMENTARY NOTATION APPROVED FOR PUBLIC RELEASE IAW AFR 190-1 ERNEST A. HAYGOOD, 1st Lt, USAF Executive Officer, Civilian Institution Programs							
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)				
FIELD	GROUP	SUB-GROUP					
19. ABSTRACT (Continue on reverse if necessary and identify by block number)							
<div style="text-align: center;">DTIC ELECTE FEB 12 1990 S D</div>							
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS				21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED			
22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF				22b. TELEPHONE (Include Area Code) (513) 255-2259		22c. OFFICE SYMBOL AFIT/CI	

**A LIBRARY OF MODULAR ROUTINES
FOR GENERATING TEST PATTERNS
FOR DIGITAL CIRCUITS**

by

THOMAS HUMBERT BELVIN, JR., B.S.

THESIS

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN ENGINEERING

Accession For	
DTIC CRA&I	<input checked="" type="checkbox"/>
DTIC TAG	<input type="checkbox"/>
DTIC AD	<input type="checkbox"/>
By	
Date	
DTIC Copy Codes	
DTIC	DTIC
A-1	



THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1988

90 02 12 036

Table of Contents

Table of Contents	iii
1. Introduction	1
2. An Overview Of Digital Testing	3
2.1 Physical Failures and Faults	3
2.2 Types Of Digital Testing	4
2.3 Approaches To Testing	4
3. Automatic Test Pattern Generation (ATPG)	6
3.1 Test Pattern Generation (TPG)	6
3.2 Automating The TPG Process	7
3.3 The Stuck-At Fault Model	7
3.4 The Single Fault Assumption	8
3.5 Path Sensitization	8
3.6 Combinational Versus Sequential Circuits	9
4. The Three Best-Known ATPG Algorithms	10
4.1 The D-Algorithm	10
4.2 The PODEM Algorithm	11
4.3 The FAN Algorithm	14
4.4 Comparing The Three Algorithms	15
4.4.1 TPG As A Search Problem	16

4.4.2	TPG Complexity	16
4.4.3	The D-Algorithm As A Search Procedure	17
4.4.4	The PODEM Algorithm As A Search Procedure	18
4.4.5	The FAN Algorithm As A Search Procedure	18
5.	Implementation Details	19
5.1	Library Overview	19
5.2	The UNIX Programming Environment	19
5.3	Program Execution	21
5.4	Data Structures	22
5.5	Constructing The Internal Representation Of The Circuit	23
5.6	Calculating Controllabilities And Observabilities	27
5.6.1	The Fan/Distance Method	27
5.6.2	The CAMELOT Method	28
5.6.3	Implementing The Calculations	30
5.7	Determining Unique Sensitisations	30
5.8	Keeping Track Of Decisions	31
5.9	The Backtrack Mechanism	32
5.10	Implementation Of The Modified D-Algorithm	33
5.10.1	The d.alg() Function	33
5.10.2	The make_pdcf() Function	34
5.10.3	The forward_implication() Function	34
5.10.4	The backward_implication() Function	35
5.10.5	The d_drive() Function	36
5.10.6	The justification() Function	37
5.10.7	The singular_cube() Function	37

5.11 Implementation Of The PODEM Algorithm	37
5.11.1 The podem() Function	38
5.11.2 The forward_implication() Function	38
5.11.3 The initial_objective() Function	38
5.11.4 The backtrack() Function	39
5.12 Implementation Of The FAN Algorithm	39
5.12.1 The fan() Function	39
5.12.2 The forward_implication() Function	40
5.12.3 The backward_implication() Function	40
5.12.4 The final_objective() Function	40
5.12.5 The fan_backtrace() Function	40
5.12.6 The unique_sensitization() Function	41
5.12.7 The justification() Function	41
6. Results	42
7. Conclusions	47
BIBLIOGRAPHY	49
Vita	

Chapter 1

Introduction

An *integrated circuit* (also called a *chip*) consists of electronic devices and their interconnections on a monolithic piece of semiconductor material. Invented in the late 1950's, the National Academy of Sciences heralded the integrated circuit as the progenitor of "The Second Industrial Revolution" [1]. The digital computers and other electronic systems which have become almost indispensable in today's society were all made possible by the integrated circuit.

Integrated circuits can be categorized into two functional groups—*analog* and *digital*. In analog circuits, signals can vary continuously over a specified range. In digital circuits, however, signals can assume only discrete values. (Today, all digital circuits use *binary* logic—signals can assume one of two possible values). Because digital circuits are capable of greater accuracy and reliability than analog circuits, they are used extensively in computational, data processing, communications, control and measurement systems [2].

By putting many devices on a single chip, *integration* reduces the cost of fabricating integrated circuits and increases their performance and reliability [3]. Until the mid-1970's, the number of electronic devices placed on a single chip nearly doubled every year (*Moore's Law* [4]). Since then, this rate of integration has decreased slightly.

In today's highly competitive semiconductor market, integrated circuit manufacturers must also be concerned with product *quality*. Both the manufacturer and the customer have vested interests in knowing: 1) the probability that a given device which leaves the production facility is functioning properly, and 2) how long a typical device can be expected to function properly while in service.

To help ensure a fabricated chip is functioning properly, manufacturers rely on *testing*. The objectives of testing are two-fold: 1) to detect manufacturing defects, and 2) to determine the cause of any defects detected so that the manufacturing process can be perfected.


By shortening the interconnections between devices and reducing the number of electrical connections which must be made between conductors, higher levels of integration have improved the reliability of integrated circuits. As the devices on a chip become smaller and more numerous, however, they also become less accessible for testing. As a result, testing becomes more difficult. Since more resources must be expended to solve more difficult problems, testing also becomes more expensive.

→ This thesis describes the development of a library of modular computer routines for the efficient and automatic generation of input patterns used to test digital integrated circuits. While solving only a small part of the testing problem, efficiently generating test patterns can lower the overall cost associated with the testing of integrated circuits.

→ next page

Chapter 2

An Overview Of Digital Testing



To help ensure an integrated circuit is functioning properly before it leaves the production facility, manufacturers (and their customers) rely on *testing*. The objectives of testing are two-fold: 1) to detect manufacturing defects, and 2) to determine the cause of any defects detected so that the manufacturing process can be perfected. (KR) ←

2.1 Physical Failures and Faults

A *physical failure* is simply a physical defect. Physical failures can occur at any time during the life of a device. When a physical failure keeps a device from meeting its functional or performance requirements, this failure results in a *fault*.

There are two basic types of faults—*logical* and *parametric*. A logical fault transforms the function realized by a circuit into some other function [5]. A parametric fault, on the other hand, alters the magnitude of a circuit parameter, causing a change in some factor such as switching time, current or voltage.

2.2 Types Of Digital Testing

In order to detect faults, digital integrated circuits are subjected to three different types of testing [5]: 1) *static* or *DC*, 2) *parametric* or *AC*, and 3) *clock-rate*.

In static testing, the device is exercised by applying binary input patterns and analyzing the steady-state outputs to determine correct functional behavior. Parametric testing is used to verify the time-related behavior of the device, as well as voltage and current levels.

Clock-rate testing is similar to static testing, except it is performed at frequencies near the maximum device operating rate. This form of testing is often used to test complex devices for which parametric testing is impractical, or to test dynamic devices with stored information which must be refreshed (e.g., dynamic metal-oxide semiconductor logic).

2.3 Approaches To Testing

There are two basic approaches to testing the logical behavior of digital integrated circuits: 1) *functional*, and 2) *structural*. Functional testing is performed to verify that a digital system accomplishes a specified task. The input patterns comprising these tests are derived manually by someone familiar with the system, and without regard to the physical structure of the circuit. Structural testing, on the other hand, is performed to ensure that the hardware is free of faults. Structural tests are generated without regard to the function realized by the circuit.

There are two basic types of functional testing—*exhaustive* and *sample*. Exhaustive functional testing of an n -input combinational circuit involves

applying all 2^n possible input combinations and comparing the outputs to the expected values. Since the number of input patterns which must be applied grows exponentially with the number of inputs, exhaustive functional testing is feasible only for circuits with few inputs.

Sample functional testing involves applying only a relatively small number of input patterns which exercise a device in its normal modes of operation. Tests generated by this approach are, however, often limited in their fault-detecting capabilities [6], and give no clue as to the cause of any detected fault.

Structural testing involves using the structural details of the circuit, along with some assumptions about the consequences of physical failures, to determine the input patterns which will distinguish between fault-free and faulty circuits. The assumptions are used to predict the effects of physical failures on the signals in the circuit, forming *fault models*. These fault models are used in conjunction with the structural details of the circuit to develop a list of all faults for which tests must be generated.

Due to its many advantages over functional testing, structural testing has become the standard approach to developing tests for digital circuits. These advantages include [7]: 1) a small number of tests can be generated for those physical failures most likely to occur, 2) the effectiveness of the resulting set of test patterns can be measured, 3) the cause of a fault can be diagnosed, and 4) test patterns can be generated algorithmically.

Chapter 3

Automatic Test Pattern Generation (ATPG)

3.1 Test Pattern Generation (TPG)

The term *test pattern generation* (TPG), or simply *test generation*, refers to the process of creating the binary input patterns for a digital circuit which, when applied to the network, will cause fault(s) to become observable at the output(s). Faults are observable when the output(s) deviate from expected values.

There are two main approaches to test generation: 1) *probabilistic*, and 2) *deterministic*. Probabilistic methods generate input patterns pseudo-randomly (i.e., without regard to circuit structure or function). Fault simulation is then used to determine which faults, if any, are detected by these patterns.

Deterministic methods select a fault from a list of faults, and attempt to generate an input pattern which will detect that fault (i.e., make the fault observable). The resulting input pattern is called a *test* for the given fault. Once a test has been generated, fault simulation is typically used to determine which other faults are detected by the same test. All detected faults are then deleted from the fault list, and the process is repeated until the fault list is empty.

3.2 Automating The TPG Process

An *algorithm* is a procedure for solving a problem in a finite number of steps. Many deterministic test generation methods have been expressed as algorithms, and can be executed either manually or by programming a computer to carry out the necessary steps. Using a computer to generate the tests is called *automatic test pattern generation* (ATPG).

The three best-known ATPG algorithms are: 1) the D-algorithm [8], 2) the path-oriented decision making algorithm (PODEM) [9], and 3) the fanout-oriented algorithm (FAN) [10]. All three use the single stuck-at fault model and the deterministic path sensitization approach to derive tests for combinational digital circuits described at the gate level (i.e., described by a logic diagram).

3.3 The Stuck-At Fault Model

The oldest, simplest, and most widely used fault model is the *stuck-at* model. Under this model, the consequences of physical failures are modeled as node signals permanently at logic 0 (*stuck-at-0*) or logic 1 (*stuck-at-1*).

Experience in industry and various studies have shown that the stuck-at fault model is still viable for today's technologies [11,12]. For example, in bipolar transistor-transistor logic (TTL), an open connection to the input of a gate can be modeled by a stuck-at-1 fault at that input. Shorts between conductors in TTL technology form wired-AND functions, and many of these physical defects can be detected by the set of tests generated for all stuck-at faults in the network [11].

Unfortunately, fewer physical defects in devices manufactured in

metal-oxide semiconductor (MOS) technology can be modeled directly by stuck-at faults. However, many of these physical defects can be detected by a set of tests generated for all stuck-at faults in the network [12].

3.4 The Single Fault Assumption

In a manufacturing environment, a single chip may have many defects. Thus it seems reasonable that multiple stuck-at faults should be considered. This greatly complicates things, however, since a circuit with n nodes has only $2n$ possible single stuck-at faults (a stuck-at-0 and stuck-at-1 for each node), but has $3^n - 1$ possible multiple stuck-at faults (3^n possible combinations, one of which is a fault-free circuit).

Fortunately, however, many studies have shown that test sets which detect a high percentage of single stuck-at faults also detect most of the possible multiple stuck-at faults [5,13,14]. Thus the single fault assumption is reasonable.

3.5 Path Sensitization

The path sensitization approach to test generation involves: 1) fault *excitation* and 2) fault *propagation*. The goal of fault excitation is to put the signal value opposite the fault value on the faulted line (i.e., a 1 for a stuck-at-0 fault, and a 0 for a stuck-at-1 fault). This creates a *fault signal* at the fault site which can then be propagated to a primary output. (The fault signal values used by Roth in the D-algorithm have become accepted—a " D " represents a 1 in the fault-free circuit and a 0 in the faulty circuit, and a " \overline{D} " represents a 0 in the fault-free circuit and a 1 in the faulty circuit).

Fault observation involves propagating fault signal(s) to primary

output(s) where they can be observed. (The path created from the fault site to the primary output where the fault is observed is called a *sensitized path*).

3.6 Combinational Versus Sequential Circuits

There are two basic types of digital circuits—*combinational* and *sequential*. The outputs of combinational circuits depend only on the values of their current inputs, while the outputs of sequential circuits depend on their past and present input values.

There are many algorithms which generate tests for combinational circuits. Test generation for sequential circuits, however, is greatly complicated by their memory elements and timing considerations. As a result, the test generation problem for sequential circuits remains largely unsolved.

To avoid the sequential test generation problem, many manufacturers have implemented their sequential circuits using various *scan-path techniques*. By placing combinational circuits between rows of latches which are accessible for testing, these techniques reduce the problem of test generation for sequential circuits to (nearly) one for combinational circuits [15,16,17].

It is even possible for physical failures to transform combinational circuits into sequential ones. For example, a short between conductors can create a feedback path, and an open pull-up transistor in a complementary metal-oxide semiconductor (CMOS) gate will fail to drive the output when it should change. These *sequential* faults require a series of input patterns to detect them, thus they may or may not be detected by a set of tests for combinational stuck-at faults.

Chapter 4

The Three Best-Known ATPG Algorithms

As mentioned previously, the three best-known ATPG algorithms are the D-algorithm [8], PODEM [9] and FAN [10]. All three of these algorithms use a deterministic path sensitization approach to test generation.

4.1 The D-Algorithm

The *D-algorithm* was the first "complete" algorithm for test pattern generation (i.e., given enough time, it will generate a test for a fault if a test exists). To be "complete", a test pattern generation algorithm must be capable of sensitizing multiple paths simultaneously.

The D-algorithm is based on the "calculus of D-cubes", where lines in the circuit can take on any one of five possible values—0, 1, X (indeterminate), D , or \overline{D} .

The *singular cover* of a gate describes the relationship between the inputs and the output. Each row of the singular cover is called a *singular cube*. The *primitive D-cubes* of a gate (also known as propagation D-cubes) describe the minimum requirements to propagate a fault signal through a gate, and are formed by *D-intersecting* the cubes in the singular cover which have different output values. The *primitive D-cubes of failure* (PDCFs) are formed by D-intersecting the singular cover of the fault-free gate with the

singular cover of the faulty gate.

The D-algorithm attempts to find a test for a fault by: 1) forming a test cube with a position for the logical value of each node in the circuit, and initializing all values to X , 2) choosing a PDCF and intersecting the test cube with this PDCF, 3) moving forward through the circuit toward the primary outputs, intersecting the test cube with the propagation D-cube for each gate encountered, and 4) when a primary output is reached, moving backward through the circuit toward the primary inputs, justifying all nodes left unjustified by the preceding D-drive operation. This last step is called the *consistency operation*, and is accomplished by intersecting the test cube with the singular cubes of gates with unjustified outputs.

If none of the propagation D-cubes for a gate are consistent with the existing test cube entries during the D-drive operation, the D-algorithm must *backtrack* to the last point where a choice existed, make an alternate selection, and try to proceed again. Conflicts occurring during the consistency operation are resolved in the same manner.

Backtracking may have to be performed repeatedly until all possible choices have been exhausted. After all possibilities have been tried without success, the D-algorithm determines that no test exists for the fault (i.e., that the fault is *undetectable* or *redundant*).

4.2 The PODEM Algorithm

The PODEM algorithm allows assignments to be made only to primary inputs. The effects of these assignments are then propagated to the internal nodes by an *implication* process, which is basically a logic simulation routine. By examining all possible primary input combinations implicitly

but exhaustively, the PODEM algorithm is also a "complete" test pattern generation algorithm.

PODEM attempts to generate a test for a given fault by repeatedly: 1) choosing an objective, and 2) determining the primary input assignments necessary to achieve that objective. This process continues until a fault signal is propagated to a primary output, or until the algorithm determines that no test exists.

The two types of objectives in PODEM are: 1) fault *excitation*, and 2) fault *propagation*. Fault excitation involves finding the primary input assignments necessary to set the value of the faulted line to the opposite of the fault value (i.e., a 1 for a stuck-at-0 fault, and a 0 for a stuck-at-1 fault). Fault propagation is the process of determining the input assignments necessary to propagate a fault signal to a primary output where it can be observed.

PODEM uses a simple but effective *backtracing* procedure to determine the input values necessary to achieve an objective. This procedure uses relative measures of the *controllabilities* of lines as inputs to a heuristic routine which determines which path to take when alternatives exist. (The *controllability* of a line denotes the relative ease with which the line can be set to a logical value.)

When determining the input values necessary to achieve an objective value at the output of a gate during the backtrace operation, two situations are possible: 1) the objective can be achieved by setting *any* of the inputs to a *controlling* state (i.e., a logic 0 for any input of an AND or NAND gate, or a logic 1 for any input to an OR or NOR gate), or 2) the objective can only be achieved by setting *all* of the inputs to *non-controlling* states (i.e., logic 1's for all inputs to AND or NAND gate, or logic 0's for all inputs to an OR

or NOR gate).

In the first case, PODEM sets the input which is *easiest* to control (i.e., has the highest controllability) to the controlling value. This strategy does not guarantee success, but when it is successful it saves a lot of time.

PODEM selects the input which is *hardest* to control as the first input to be set to a non-controlling value in the second case. This strategy saves time in the long run by allowing the earliest possible determination that an objective cannot be achieved.

The backtrace operation proceeds gate by gate towards the primary inputs until an input is finally reached. The value determined by the backtrace routine is then assigned to the input, and the implication operation is performed to propagate the effects to the internal nodes. If a fault signal reaches a primary output during the implication process, a test has been generated. Otherwise, a check is made to see if the current objective has been achieved. If it has, then a new objective is chosen. If the current objective has not been achieved, the backtrace operation is repeated.

During the fault propagation phase, PODEM selects as its next objective to propagate a fault signal through a gate with an output which is both: 1) closest to a primary output, and 2) from which it is still possible to sensitize a path to a primary output. PODEM uses a simple distance measure of observability for the first requirement, and an "X_path_check" routine for the second. The X_path_check routine searches for a path from the output of a gate to a primary output along which all values have not yet been determined (i.e., along which all logic values are "X").

Each assignment made to the primary inputs, called *decisions*, are stored in a last-in first-out (LIFO) stack. Associated with each decision is a

"retry" flag which is set when the originally assigned value is reversed during a backtrack operation.

When a conflict is detected by 1) attempting to reassign a value to a primary input opposite its current value, or 2) when the *X_path_check* routine fails for all gates with indeterminate outputs and fault signal(s) on their inputs (i.e., all gates on the *D-frontier*), PODEM must execute the backtrack operation.

The PODEM backtrack operation removes decisions from the decision stack until a decision which has not yet been retried is found. The values of any previously retried decisions (i.e., decisions with their retry flags set) are returned to "X". The value of the first decision removed from the stack which has not been retried, however, is reversed, its retry flag is set, and it is added to the top of the stack. The implication routine then propagates the effects of all decisions on the decision stack to the internal nodes, and processing continues.

If the decision stack becomes empty during the backtrack operation (i.e., no decision is found which has not previously been retried), then PODEM determines that no test is possible for the fault.

4.3 The FAN Algorithm

The FAN algorithm is a refinement of the PODEM algorithm, including more effective deterministic and heuristic measures. As its name implies, FAN pays special attention to the fanout points in a logic circuit.

Fan allows assignments to: 1) *head lines*, and 2) *fanout stems*. A head line is a *free* line which feeds a *bound* line. A *bound* line is reachable from

some fanout point. A line which is not a *bound* line is a *free* line. Assignments are kept in a LIFO stack just as they are in PODEM.

Like PODEM, FAN uses a backtracing procedure to satisfy objectives. However, instead of backtracing along a single path, FAN backtraces along multiple paths simultaneously. As it moves backward through the circuit towards the primary inputs, the FAN backtrace routine resolves conflicting requirements at fanout points, and stops at head lines. (The backtrace operation can be stopped at head lines since the primary input assignments necessary to justify head line assignments can always be found without backtracking.)

When conflicting requirements occur during the backtrace operation at fanout stems which are not reachable from the fault site, FAN uses a weighting scheme to determine which binary value (0 or 1) to assign to this node. (Since binary values are assigned, these points must not be reachable from the fault site). Resolving inconsistencies as early as possible is one of the keys to FAN's more efficient operation.

As in PODEM, if the decision stack becomes empty during the back-track operation, the FAN algorithm determines that no test is possible.

4.4 Comparing The Three Algorithms

In order to compare the three algorithms, it is necessary to put the TPG problem in perspective.

4.4.1 TPG As A Search Problem

TPG is essentially a search procedure—a search for the primary input assignments which will make a fault observable if it exists. Search procedures can be categorized as *blind* or *guided*. In a blind search, the order in which alternatives are explored is unaffected by the goal criteria or the nature of the unexplored portion of the graph. Examples of blind searches are *depth-first* and *breadth-first* [18].

Guided searches, on the other hand, use rules of thumb garnered from a working knowledge of the problem domain and the nature of the goal to help direct the search in more promising directions—*heuristics*. Guided searches are usually more efficient than blind searches, but may result in the same exhaustive enumeration in the worst case. Examples of guided searches are *hill-climbing* and *best-first* [18].

4.4.2 TPG Complexity

The TPG process for a single stuck-at fault in a general combinational network has been shown to be NP-complete [20], which puts TPG in the same complexity category as the well-known *traveling salesman* and *knap-sack* problems. All known algorithms for NP-complete problems require an *exponential* amount of time in the *worst case*. (Exponential-time algorithms use impractically large amounts of computation time even for relatively small problem sizes).

While no one has been able to *prove* that NP-complete problems can be solved in “reasonable” amounts of time, guided search techniques are often able to do just that.

4.4.3 The D-Algorithm As A Search Procedure

The D-algorithm is essentially a blind search, using only basic information about the configuration of the circuit and the logical functions of the elements to generate tests.

The D-algorithm also suffers from a large search space—all the nodes in the circuit. Since all known algorithms for NP-complete problems like TPG require an exponential amount of time in the worst case, the upper bound for the time required by the D-algorithm is relatively high. To make matters worse, this search space contains nodes which are *interrelated*—assigning a value to one node often conflicts with the assignments of other nodes. Propagating fault signals to primary outputs before attempting to justify assignments made along the way delays the recognition of these conflicts, and often results in wasted time and effort.

Realizing that the original D-algorithm is rather inefficient, two major improvements were made in the version implemented for this thesis. These modifications are conceptually the same as those made by Roth in his improved version of the D-algorithm [19]: 1) forward and backward implication operations were added to carry the uniquely implied effects of a decision throughout the circuit immediately following the decision, and 2) the observabilities of lines in the circuit were used to decide which fault signal to propagate next. Performing the implication operations allow conflicts to be detected earlier, reducing the time needed to generate tests by reducing the number of backtracks. Always propagating fault signals closest to the primary outputs often results in generating tests in less time as well.

4.4.4 The PODEM Algorithm As A Search Procedure

The search space for the PODEM algorithm consists of a small subset of all the nodes in a circuit—the primary inputs. This means that the upper bound on the time required by PODEM is much less than that of the D-algorithm. Even more important is the fact that the nodes in this search space are *independent*—assigning a value to one node can never conflict with assignments at other nodes.

PODEM is also a guided search procedure, using heuristics to determine which path to take during the backtrace operation, and which gate to propagate a fault signal through during the fault propagation phase.

4.4.5 The FAN Algorithm As A Search Procedure

As mentioned previously, the FAN algorithm is a refinement of the PODEM algorithm which includes more effective deterministic and heuristic measures. The search space for FAN consists of another small subset of all nodes in the circuit—head lines and fanout points. Thus the upper bound on the time required by FAN is also much lower than that of the D-algorithm. Fanout stems which are not reachable from the fault site are included in the search space in an effort to quickly resolve conflicting requirements at these points.

FAN also includes a number of deterministic measures which help to identify conflicts at an earlier stage, thus reducing the total time and number of backtracks required.

Chapter 5

Implementation Details

5.1 Library Overview

The functions used to implement the D-algorithm, PODEM and FAN were coded in the C programming language [21]. These implementations contain about 13,000 lines of C code in over 150 functions in 60 different files. The files are compiled separately, then linked together to form the programs which implement the above algorithms.

As shown in Figure 5.1, all implementations share the same data input routines as well as the forward implication operation. PODEM and FAN share the same measures of controllability and observability, as well as the X_path_check function discussed previously. D-algorithm and FAN share the backward implication and justification operations.

5.2 The UNIX Programming Environment

All of the functions used to implement the D-algorithm, PODEM and FAN were developed in a UNIX¹ operating system environment. Many of the attributes of C and UNIX made it possible to develop the library in a relatively short period of time, and with little prior knowledge of the language

¹UNIX is a Trademark of Bell Laboratories.

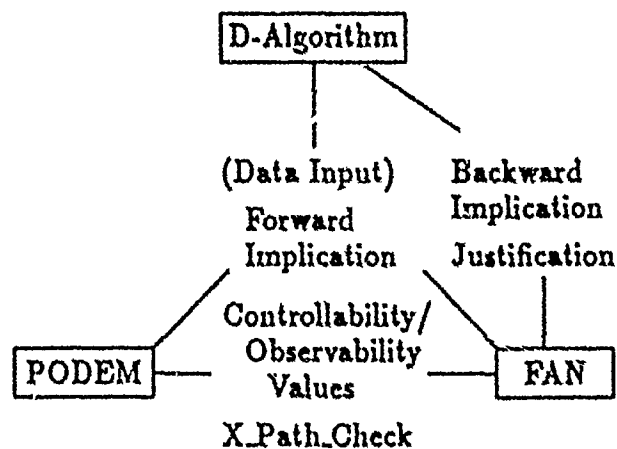


Figure 5.1: Routines Shared By The Three Best-Known ATPG Algorithms.

or of the operating system. They include:

- The functional modularity of the C programming language.
- Separate compilation of multiple source files.
- The UNIX make utility.
- A powerful C debugger dbx.
- Two other UNIX utilities lex and yacc.

C programs consist of one or more functions in one or more files. If a C program consists of two or more files, these files are compiled independently, then connected together to form a single executable program by the *linker*. Thus, by its very nature, C supports modular programming and libraries of functions.

One UNIX utility which proved invaluable was the make utility. Make is an automatic program compilation utility which help maintain large programs which have been broken up into several files. When executed, make

checks to see if any of the files needed to create the target executable file have been modified since the last compilation. If so, these files (and/or any files which depend on them) are automatically recompiled and linked together to form an up-to-date version of the program. This action saves programmers considerable time and effort in a software development environment.

Any large software development project also requires a powerful debugger like dbx. For example, since C has no automatic array bounds checking, overwriting the bounds of an array can (and did) cause baffling run-time errors which are almost impossible to diagnose in a large program without a sophisticated debugger.

Two UNIX utilities called lex (lexical analyzer generator) and yacc (yet another compiler-compiler) made it very easy to input data from separate data files. These utilities free programmers from much of the coding required to handle data entry, allowing them to concentrate their efforts on their particular application.

5.3 Program Execution

All files are compiled and linked together to form an executable program called "dalg". The file names "podem" and "fan" are then linked to the "dalg" file. When the user invokes the program with one of these names, the first argument on the command line determines which implementation is executed (i.e., "dalg" for the D-algorithm, "podem" for PODEM, or "fan" for the implementation of FAN).

The second argument on the command line is the name of the input file containing the circuit description and the list of target faults. For example, typing "dalg our.data" will execute the modified version of the D-algorithm

using the information in the file "our.data".

Run-time statistics (test generation time, number of backtracks, etc.) are automatically appended to the file "dalg.D" (for the D-algorithm), "podem.P" (for PODEM) or "fan.F" (for FAN) in the subdirectory "Data". Flags in the file "globals.h" determine whether the test patterns are printed to the screen, to a separate file, or both. When the flag indicating data is to be written to a separate file is set, the test patterns generated are written to a unique file in a subdirectory called "Fault_Info". If test generation for a given fault is discontinued when a preset backtrack limit is exceeded (i.e., the fault is "aborted") or the fault is determined to be redundant, the appropriate message is also written to this file.

5.4 Data Structures

The implementations of D-algorithm, PODEM and FAN all share common data structures. The internal representation of the circuit is *line-oriented* and consists of collections of basic data structure elements joined by pointer variables to form a linear linked list.

Figure 5.2 shows a very simple circuit and its internal representation. The information about each line is contained in two basic structures linked by pointers: 1) the *module* structure, and 2) the *gate_info* structure. (In Figure 5.2(b), the module structures are the large boxes across the top labeled "module #n", and the gate_info structures are directly under the module structures). Each module structure contains: 1) the line number, 2) the code for the logical value of the line (0, 1, X , D or \overline{D}), 3) the controllability and observability values, 4) various flags used to speed up execution, 5) a pointer to the associated gate_info structure, and 6) a pointer to the next module in

the list.

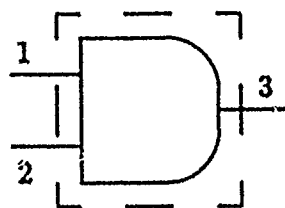
Each *gate_info* structure contains: 1) a code for the type of gate the line originates from, 2) a pointer to a linked list of *predecessors* (i.e., the inputs to the gate the line originates from), 3) a pointer to a linked list of *successors* (i.e., either the output of the gate the line is an input to, or, if the line is a fanout stem, the line numbers of its branches), and 4) a pointer to a linked list of *unique sensitizations* used by the FAN algorithm.

It was not clear at the beginning what information needed to be stored for efficient implementations—the data structures evolved as the development progressed and the test circuits grew in size and complexity. The extensive use of pointers and flags to store frequently needed information helped increase the efficiencies of the implementations by eliminating duplication of effort. (Pointers and flags can be set once during preprocessing, or at the start of the search for a test for a new fault, and functions needing this information later have it immediately available).

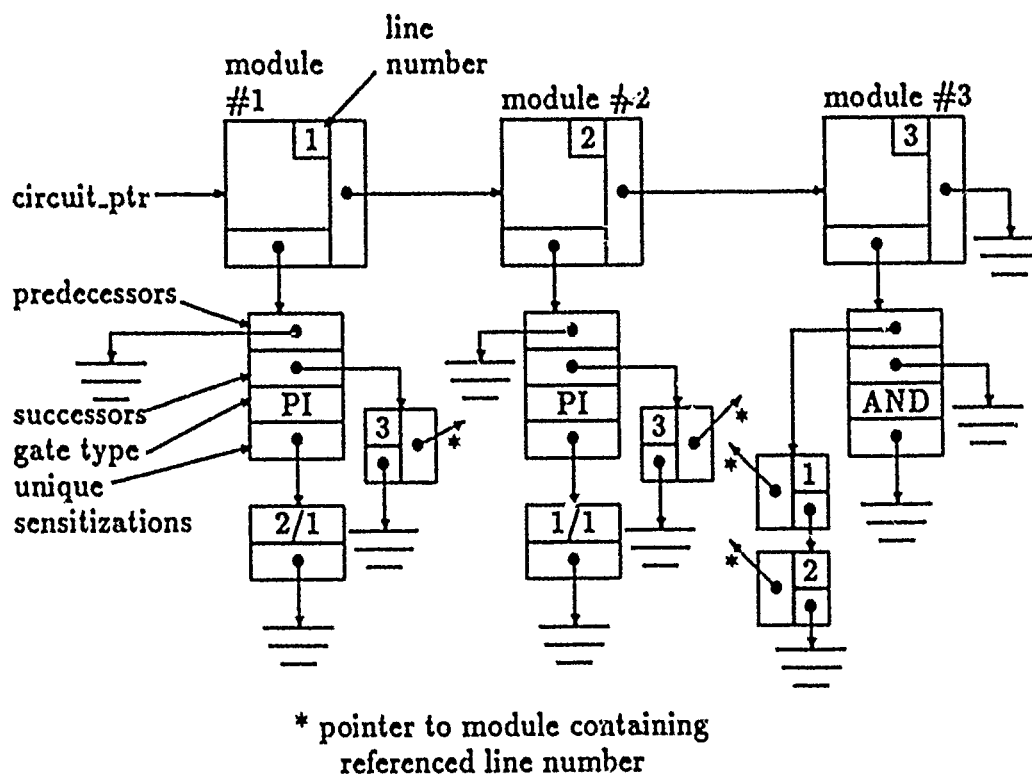
For example, elements in the linked lists of predecessors and successors must have pointers to the lines they reference. Supplying only the line numbers of successors or predecessors would result in slow and repeated searching in order to access their information. Also, many functions need to know if a given line is a primary output, and setting a flag in the module containing the information about the line can answer this question quickly.

5.5 Constructing The Internal Representation Of The Circuit

During preprocessing, information from the input file is used to build an internal representation of the circuit. A program created by the lex utility

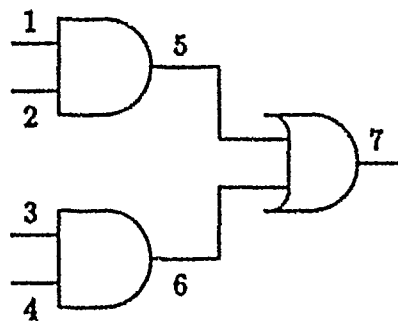


(a) Example Circuit.



(b) Internal Representation Of (a).

Figure 5.2: An Example Circuit And Its Internal Representation.



(a) Example Circuit.

```

/* Example Circuit */
1 2 and 5;      /* gates, inputs and outputs */
3 4 and 6;
5 6 or 7;
$              /* end of section */
po 7;         /* primary output list */
$
1 sa0; 6 sa1;  /* fault list */
$$            /* end of input */

```

(b) Input File For (a).

Figure 5.3: Example Circuit and the Corresponding Input File.

feeds information from the input file to a yacc-created program in recognized character sequences called *tokens*. The program created by the yacc utility compares the order in which these tokens are received to predefined rules for input file organization. When one of these rules is matched, a user-defined *action* is invoked. These actions construct the linked list of data structures which constitutes the internal representation of the circuit. Figure 5.3 shows an example circuit and the corresponding input file.

The types of gates recognized include ANDs, ORs, NANDs, NORs, EXCLUSIVE-ORs and INVERTERS. (The INVERTER is modeled internally

as a single-input NAND gate). Line numbers must be positive integers, and the line numbers of gate outputs must be greater than any of the line numbers of their inputs. This aids in checking the input file for errors resulting in feedback.

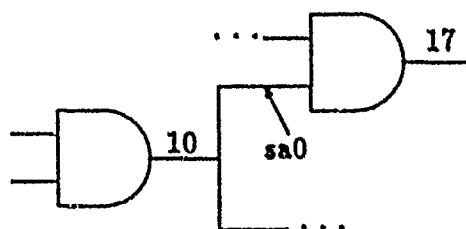
After all gates and their inputs and outputs have been specified, the primary outputs are listed. This information is needed to identify primary outputs which are fanout branches, and also helps in checking the input file for errors. (Every line in the input file must be a gate input, be identified as a primary output, or both).

Following the list of primary outputs is the list of target faults. Target faults can be specified explicitly, or all possible faults can be specified by entering the word "all". When target faults are specified explicitly, this information is stored in a linked list.

It was decided early on not to burden the user with specifying each and every fanout branch in the circuit. Data structures for fanout branches are generated *automatically* by a special routine after the basic circuit representation has been constructed. Unique line numbers are assigned to the fanout branches by: 1) multiplying the line number of the stem by 100, 2) adding an integer representing the branch number, and 3) making the result negative. (The value 100 allows for a maximum of 99 fanout branches per stem).

Relieving the user of the requirement to specify all fanout branches has one drawback—specifying faults on fanout stems requires a special format and extra data handling. Figure 5.4 shows how a fault on a fanout branch is specified.

The routine which adds the data structures for fanout branches also



(a) Faulted Fanout Branch.

...; 10 sa0 17; ...

(b) Fault Specification For (a).

Figure 5.4: Fault Specification For A Faulted Fanout Branch.

updates the list of primary outputs and the list of target faults after line numbers have been assigned to the fanout lines.

5.6 Calculating Controllabilities And Observabilities

As mentioned previously, both PODEM and FAN use measures of controllability and observability as inputs to heuristic routines which decide which path to take when alternatives exist. Two measures of controllability and observability were implemented [22]: 1) *fan/distance* and 2) *CAMELOT*, which stands for computer-aided measure for logic testing.

5.6.1 The Fan/Distance Method

Using the fan/distance method, the controllability of a line is calculated by: 1) assigning each node an intrinsic controllability equal to its

fanout minus one, and 2) working from primary inputs to outputs, adding the sum of the input controllabilities to the intrinsic controllability values of gate output lines. Using this method, lines with *lower* controllability values are *easier* to control.

The observability of each line is calculated by: 1) assigning a zero value to primary outputs, and 2) working backwards through the circuit from primary outputs to inputs, gate input lines are assigned observability values equal to the output observabilities plus one. Using this method, lines with *lower* observability values are *easier* to observe.

5.6.2 The CAMELOT Method

The CAMELOT method calculates the controllabilities of lines by: 1) assigning the value one to primary inputs, and 2) working from the primary inputs to the outputs, the controllabilities of gate outputs are calculated as follows:

N_0 = total number of possible input patterns
producing a zero at the output

N_1 = total number of possible input patterns
producing a one at the output

CTF = controllability transfer function
 $= 1 - (N_0 - N_1)/(N_0 + N_1)$

SIC = sum of the input controllabilities

TNI = total number of gate inputs

GGO = controllability of the gate output
 $= CTF \cdot (SIC/TNI)$

Using the CAMELOT method, controllability values are real numbers between zero and one, and lines with *higher* values are *easier* to control.

The observability of each line is calculated as follows: 1) the observabilities of the primary outputs are assigned the value one, and 2) working from the primary outputs back towards the inputs, the observabilities of gate inputs are calculated:

N_P = the total number of ways to propagate
a fault signal on this line

N_D = the total number of ways to block propagation
of a fault signal on the line

OTF = observability transfer function
 $= N_P / (N_P + N_D)$

OGO = observability of the gate output

$SCOI$ = sum of the controllabilities
of the other gate inputs

$TNOI$ = total number of other inputs

OGI = observability of the gate input
 $= OTF \cdot OGO \cdot (SCOI / TNOI)$

At fanout points,

OB_i = observability of fanout branch "*i*"

OS = observability of the stem
 $= 1 - \prod_{i=1}^n (1 - OB_i)$

The CAMELOT observability values are also real numbers between zero and one, and lines with *higher* values are *easier* to observe.

The fan/distance method was implemented first, so the CAMELOT values were made compatible by rounding their inverses to the nearest integer. This worked well for the controllability values, but the converted observability values began to exceed the maximum value which could be represented by the 32-bit signed integer format of our computer. The problem was solved by converting the six most significant bits of the format (after the sign bit) to an exponent. This alleviated the overflow problem, and still allowed for direct comparison of the magnitudes of observability values.

5.6.3 Implementing The Calculations

Controllabilities and observabilities are computed using the same general method. First, the values of the primary inputs (for controllability) or the primary outputs (for observability) are set to one, and the values for all other lines are initialized to minus one, indicating they have not yet been computed. All final (non-negative) values are then used to calculate the values for their successors (for controllability) or their predecessors (for observability) during repeated passes through the internal representation of the circuit.

Passes continue to be made through the data structures until: 1) all values have been determined (i.e., all values are positive), or 2) a pass through the data structures fails to determine at least one previously undetermined value. In the latter case, an error is reported, and processing stops.

5.7 Determining Unique Sensitizations

The *unique sensitizations* of a line are the path sensitization values common to all possible paths from the line to a primary output. As with the

calculations of line observabilities, the unique sensitizations of all lines are determined by the unique sensitizations of their successors during repeated passes through the internal representation of the circuit.

On the first pass through the data structures, the pointers to the linked lists of unique sensitizations are set to NULL (i.e., zero or none) for the primary outputs, and to a predetermined value which signifies they have not yet been determined for all other lines. (The address of the module data structure itself was used).

The unique sensitizations of a line are determined once the unique sensitizations of its successor(s) have been found. If the line is not a fanout stem, then it has only one successor. Its unique sensitizations can be determined using: 1) the type of gate the line is an input to, 2) the line numbers of the other inputs to that same gate, and 3) the unique sensitizations of its successor (i.e., the unique sensitizations of the gate output line).

If, on the other hand, the line is a fanout stem, then its unique sensitizations are the sensitizations common to the sets of unique sensitizations of its fanout branches.

As with the controllability and observability calculations, passes continue to be made through the data structures until: 1) all unique sensitizations have been determined, or 2) a pass through the data structures fails to determine the unique sensitizations for at least one new line. Again, in the latter case, an error is reported and processing stops.

5.8 Keeping Track Of Decisions

All three algorithms use a LIFO stack to keep track of decisions and ensure all possible choices are examined, either explicitly or implicitly. When

a new decision is made, it is pushed onto the top of the stack.

Along with the decision itself, the implemented version of the D-algorithm (which includes forward and backward implication operations) also saves: 1) the PDCF (for the first decision) or the propagation D-cube (for subsequent decisions) selected, 2) the backward implications, and 3) the resulting D-frontier. This information is used to restore the state of the circuit when a backtrack occurs.

If the implications of a new decision do not conflict with previous decisions and their effects, the FAN algorithm saves this decision and: 1) its backward implications, 2) any unique sensitization values used, and 3) the backward implications of any unique sensitizations used.

Both the D-algorithm and FAN also have a second decision stack for the justification operation. Along with the singular cubes selected, the effects of backward implications must again be stored. (Using forward and backward implication in the justification operation also reduces the amount of time required to generate a test).

The PODEM algorithm is the only one of the three which saves only the decisions on its decision stack.

5.9 The Backtrack Mechanism

When a conflict occurs in any of the three algorithms, they must *backtrack* to the last decision which involved making a choice and try another alternative. In order to accomplish this action, the circuit must be returned to the state which existed when the choice was made. The D-algorithm and FAN save the state of the circuit, while PODEM is able to propagate the effects of the previous decisions to restore the state.

The section which describes how decisions are recorded also describes what information is saved by the D-algorithm and FAN in order to restore the circuit state.

5.10 Implementation Of The Modified D-Algorithm

As mentioned previously, two improvements to the original D-algorithm were incorporated into the version implemented: 1) forward and backward implication operations following each decision, and 2) the observabilities of lines were used to decide which fault signal to propagate next.

The implementation of the D-algorithm includes the following major functions:

- `d_alg()`
- `make_pdcf()`
- `forward_implication()`
- `backward_implication()`
- `d_drive()`
- `justification()`
- `singular_cube()`

5.10.1 The `d_alg()` Function

The `d_alg()` function orchestrates the operations of the other functions. It also handles the decision stack and backtrack operations discussed previously.

5.10.2 The `make_pdcf()` Function

The `make_pdcf()` function generates a linked list of all primary D-cubes of failure (PDCFs) using: 1) the type of gate, 2) the number of gate inputs, and 3) the type of fault (stuck-at-0 or stuck-at-1) at the output of the gate. Each list element contains the coded values for the input(s) and output corresponding to a single PDCF. (Each output is either D or \overline{D}).

5.10.3 The `forward_implication()` Function

The `forward_implication()` function, used by all three algorithms, transmits the logical effects of changing values through the circuit in the forward direction (i.e., towards the primary outputs). Effects are propagated when the output value of a gate can be determined by its input values.

When a decision is made, the value of the decision node is changed, and the decision is examined for any forward implications. The successor(s) of the decision node are automatically added to the beginning of a circular buffer of lines to be evaluated. The forward implications are determined by successively: 1) removing a line from the buffer, 2) determining its new value from the values of its predecessors, and 3) comparing this new value to the current value. If the current value is X , the new value is assigned, and the successor(s) of the line are added to the end of the buffer. If the new value is the same as the old value, no further action is taken, and processing continues with the next line in the buffer. (If the current output was anything but X , and the new value is not the same as the old value, a conflict occurs.) Evaluations continue to be carried out until: 1) the buffer becomes empty, or 2) a conflict occurs.

A circular buffer is used instead of a linked list to avoid the overhead

associated with linked list operations:

- Creating new list elements (dynamic memory allocation)
- Adding and deleting elements from a list
- Returning unused elements (i.e., memory) to the system

The size of the buffer may be varied, and may well become a compromise between the desired speed of execution and the size of the available memory. A linked list is used to handle any overflow from the buffer. As space becomes available in the buffer, information in the overflow list is moved into the buffer.

If a conflict occurs, the `forward_implication()` function is called a second time to restore the state of the circuit to its condition before the last decision was made. This is accomplished by changing the decision node back to X , and propagating the effects until lines tagged as having values determined by previous decisions are encountered.

Each time a fault signal is propagated, the linked list containing the lines in the D-frontier is updated. This list is also updated each time a value which allowed the propagation of a fault signal is changed back to X following a conflict.

5.10.4 The `backward_implication()` Function

The `backward_implication()` function, also used by the FAN algorithm, transmits uniquely implied logic values backward through the circuit towards the primary inputs. Values are uniquely implied when there is only one set of input values which results in the present output value.

Each time a decision is made, it is examined for backward implications before the `forward_implication()` function is performed. (PODEM

makes assignments only to primary inputs, thus there can be no backward implications.) If a predecessor value is uniquely implied, its value is changed, and this change is recorded in one of two linked lists. If the predecessor is not a fanout node, the change is recorded in the "backward impact" list. If the predecessor node is a fanout node, the change is recorded in a linked list of "forward impacts." (Backward implication is always performed before forward implication, and the calling routine must also send those lines in the "forward impact" list to the forward implication routine.)

These lists, containing the results of the backward implication process, are used later to restore the state of the circuit during a backtrack. (See the section describing the backtrack operation).

5.10.5 The d_drive() Function

The d_drive() function automatically generates the propagation D-cubes for all gate types except EXCLUSIVE-OR gates by assigning the binary values (0 or 1) to input lines which have not yet not yet been determined (i.e., are still *X*) in order to propagate fault signals toward primary outputs. The advance_by_X() function generates the propagation D-cubes for EXCLUSIVE-OR gates.

The d_drive() function continues to propagate fault signals toward primary outputs until: 1) a fault signal arrives at an output, or 2) all possible ways to propagate fault signals have been tried without success. In the latter case, the D-algorithm determines that no test exists for the fault.

5.10.6 The justification() Function

The justification() function, also used by the FAN algorithm, uses information supplied by the singular_cube() function to find non-conflicting assignments which justify all unjustified lines. Processing takes place in a doubly linked list originally containing a list of all unjustified lines. This list is expanded to include the line numbers of all predecessors of these unjustified lines as well.

After a singular cube is selected and the values of affected lines are changed, the forward and backward implications of these assignments are examined. If conflicts occur, new singular cubes are chosen, and their implications are propagated forward and backward through the circuit. This process continues until: 1) all lines are justified, or 2) all possible combinations of singular cubes have been tried without success. In the latter case, the assignments resulting from the justification() routine are reversed, and the calling routine is informed of the failure.

5.10.7 The singular_cube() Function

The singular_cube() function generates the singular cubes for all gates on an as needed basis. To determine the next singular cube to be generated and returned, it needs to know: 1) the gate type, 2) the number of inputs, and 3) the number assigned to the last singular cube generated.

5.11 Implementation Of The PODEM Algorithm

The implementation of the PODEM algorithm includes the following major functions:

- `podem()`
- `forward_implication()`
- `initial_objective()`
- `backtrace()`

5.11.1 The `podem()` Function

The `podem()` function orchestrates the operations of the other functions. It also handles the decision stack and backtrack operations discussed previously.

5.11.2 The `forward_implication()` Function

The `forward_implication()` function, shared by all three implementations, is discussed in the preceding section on the modified D-algorithm.

5.11.3 The `initial_objective()` Function

The `initial_objective()` function is used to generate a fault signal at a fault site and to propagate fault signals to primary outputs. A fault signal is established by achieving the objective of setting the value of the faulted line to the opposite of the fault value (i.e., a 0 for a line with a stuck-at-1 fault, and a 1 for a line with a stuck-at-0 fault). This objective is passed to the backtrack routine.

When propagating fault signals toward the primary outputs, the objective becomes propagating a fault signal through the gate on the D-frontier with an output that has both: 1) the highest observability, and 2) an X_path to a primary output. This objective is also passed to the backtrack

routine.

5.11.4 The backtrace() Function

The backtrace() routine uses the heuristics discussed previously to move backward through the circuit in an effort to determine a primary input assignment which will achieve the initial objective.

5.12 Implementation Of The FAN Algorithm

The implementation of the FAN algorithm includes the following major functions:

- fan()
- forward_implication()
- backward_implication()
- final_objective()
- fan_backtrace()
- unique_sensitization()
- justification()

5.12.1 The fan() Function

The fan() function orchestrates the operations of the other functions. It also handles the decision stack and backtrack operations discussed previously.

5.12.2 The forward_implication() Function

The forward_implication() function, common to all three implementations, is discussed in the preceeding section on the modified D-algorithm.

5.12.3 The backward_implication() Function

The backward_implication() operation is shared by the implementations of the D-algorithm and FAN, and is discussed in the preceeding section on the modified D-algorithm.

5.12.4 The final_objective() Function

The final_objective() function: 1) assigns the appropriate binary value to the head line or fanout point objective returned by the det_final_obj() routine, and 2) returns a pointer to the module data structure containing the information about the objective line to the calling fan() function. The det_final_obj() operation uses the status of the backtrace flag and the sets of head line and fanout point objectives, formed by the fan_backtrace() function, to determine the final objective.

5.12.5 The fan_backtrace() Function

The fan_backtrace() function moves backward through the circuit along multiple paths in a *breadth-first* manner from the initial objective lines towards the primary inputs, resolving conflicting requirements at fanout points which are *not* reachable from the fault site, and stopping at head lines. (Not being reachable from the fault site guarantees that a binary value can be assigned to the fanout stem).

To implement the multiple backtrace in a breadth-first manner, objectives are removed from the *head* of a linked list of current objectives, and, if they are not head lines or fanout branches, heuristic rules [10] are used to determine the next objectives. These next objectives are then added to the *tail* of the list of current objectives, but only if a search of the list verifies they are not already included.

5.12.6 The `unique_sensitization()` Function

The `unique_sensitization()` function is executed when there is only a single gate in the D-frontier which has an *X_path* to a primary output.

The required assignments, determined during preprocessing, are made in an effort to reduce the time needed to generate a test. (See the earlier section on how these unique sensitizations are determined).

5.12.7 The `justification()` Function

After a fault signal has been propagated to a primary output, the `justification()` function is used to justify the assignments at head lines and fanout points. The `justification()` function, shared by the D-algorithm and FAN, is discussed in the earlier section on the modified D-algorithm.

Chapter 6

Results

All three implementations were used to generate tests for a standard set of circuits known as the "ISCAS circuits" [23]. The characteristics of these circuits [24] are shown in Table 6.1.

Table 6.2 shows the fault coverage, the average number of backtracks, and the total time required to generate complete sets of tests for nine of the ISCAS circuits using the implemented version of the D-algorithm. Tables 6.3 and 6.4 show the same information for the PODEM and FAN implementations respectively. All results were obtained with a backtrack limit of 10 and without fault simulation during test generation to find any other faults detected by a test for a particular fault. Fault simulation was used later, however, to verify that the tests produced did indeed detect the faults for which they were generated.

The fault coverages shown are the fault coverages for collapsed sets of all faults. Collapsing was performed using equivalence relationships only, and typically resulted in sets of target faults half the size of the original sets of all possible stuck-at faults. The numbers in parentheses beside the fault coverage figures are the percentages of faults in the collapsed sets which were determined to be redundant. The fault coverage figures do not include these redundant faults.

Table 6.1: Characteristics Of The ISCAS Circuits

Circuit Name	Total Lines	Total Gates	Input Lines	Output Lines	Average Fanout
c432	432	160	36	7	2.65
c499	499	202	41	32	4.34
c880	880	383	60	26	3.50
c1355	1355	546	41	32	2.97
c1908	1908	880	33	25	2.58
c2670	2670	1193	233	140	2.74
c3540	3540	1669	50	22	3.15
c5315	5315	2307	178	123	3.51
c6288	6288	2406	32	32	2.64
c7552	7552	3512	207	108	2.95

Table 6.2: Results For Implementation Of D-Algorithm

ISCAS Circuit	Fault Coverage	Average Back-tracks	Time (sec.)
c432	98.3(0.2)	0.438	39
c499	94.5(1.1)	0.937	85
c880	100	0.048	42.3
c1355	40.4(0.5)	7.604	434
c1908	93.3(0.4)	0.872	561
c2670	88.3(3.0)	1.446	432
c3540	89.8(3.7)	1.222	1055
c5315	94.8(1.1)	0.802	910
c7552	81.9(0.7)	2.540	2892

Table 6.3: Results For Implementation Of PODEM

ISCAS Circuit	Fault Coverage	Average Back- tracks	Time (sec.)
c432	98.5(0)	1.418	28
c499	95.8(0)	1.241	85
c880	100	0.007	33
c1355	91.4(0)	1.135	399
c1908	98.9(0.3)	0.123	261
c2670	91.7(0.9)	1.092	365
c3540	89.6(1.0)	1.306	1160
c5315	96.8(0.8)	0.513	873
c7552	89.8(0.7)	1.682	3574

Table 6.4: Results For Implementation Of FAN

ISCAS Circuit	Fault Coverage	Average Back- tracks	Time (sec.)
c432	98.7(0.2)	0.447	171
c499	79.2(1.1)	3.074	284
c880	100	0.781	133
c1355	49.3(0.5)	7.187	797
c1908	88.5(0.4)	2.122	1778
c2670	89.3(3.5)	1.285	1306
c3540	87.2(3.7)	1.806	3446
c5315	97.5(1.1)	0.712	2069
c7552	86.3(0.8)	2.936	9659

The average number of backtracks is calculated by dividing the total number of backtracks by the number of faults in the collapsed set.

The time recorded in the tables is the time actually spent in the test generation routines, and does not include the time required for preprocessing (i.e., the time required to build the data structures, calculate the controllabilities and observabilities, etc.).

With a backtrack limit of 10, the results show that the PODEM implementation often outperformed the other two implementations in all three categories. What the results do not show is the effect redundant faults have on these measurements when the backtrack limit is raised in an effort to improve fault coverage. The FAN and D-algorithm implementations frequently determine that faults are redundant in fewer backtracks than the PODEM implementation, suggesting that their relative performances may improve for large circuits when the backtrack limit is high.

The same statistics for a previous implementation of PODEM [25] considered reasonably efficient is shown in Table 6.5. Comparing this data with the results in Table 6.3 suggests that the new implementation is at least as efficient as the previous one. The greatest differences occur in the average number of backtracks, and are most likely due to the different guidance heuristics used.

Software profiling, used to gather run-time statistics on the program functions, also yielded some interesting results. The implemented version of the D-algorithm typically spends most of its time in the forward implication, backward implication and justification operations. The PODEM implementation typically spends most of its time in the forward implication, backtrace and the main control operations. The implementation of FAN typically

Table 6.5: Results For Previous Implementation Of PODEM

ISCAS Circuit	Fault Coverage	Average Back- tracks	Time (sec.)
c432	93.3(0)	0.971	30
c499	97.6(0)	0.325	83
c1355	96.8(0)	0.881	703
c1908	99.1(0.3)	0.141	304
c2670	95.6(0.9)	0.471	427
c3540	90.2(1.0)	1.120	1205
c5315	98.2(0.8)	0.196	1161
c7552	97.1(0.7)	0.429	3365

spends the bulk of its time in the forward implication, multiple backtrace and various linked list operations.

Chapter 7

Conclusions

A library of modular computer routines for ATPG was developed and used to implement the three best-known ATPG algorithms—the (modified) D-algorithm, PODEM and FAN. These implementations efficiently generate tests for digital circuits, and can be used to reduce the escalating costs associated with the testing of digital integrated circuits.

In addition, this library approach facilitates the implementation of new ATPG algorithms, and also provides a favorable environment in which the relative performances of ATPG algorithms can be compared. New ATPG algorithms can be implemented by combining the existing routines in new and different ways, adding new routines, substituting new routines for existing ones, etc. Hybrid test generation programs which switch from one implementation to another during execution can also be developed and optimized.

In the library environment, the sharing of common functions tends to nullify any differences in performance due to implementation, thereby accentuating differences in the efficiencies of the underlying algorithms. For example, all three of the algorithms implemented spend most of their time in the forward implication routine. Thus their relative performances could be greatly affected by the efficiencies of their forward implication operations. By sharing a common forward implication routine, however, any differences

in their relative performances are more likely to result from differences in the efficiencies of their basic algorithms.

The hybrid test generation program approach warrants further study. For example, if the fault coverage provided by the PODEM implementation with a low backtrack limit is not sufficient, it may be better to attack the aborted faults with the FAN or modified D-algorithm implementation rather than simply trying again with a higher backtrack limit. Through further experimentation and analysis, the best overall strategy for combining the three implementations to form one test generation program can be determined.

The main drawback to the library approach is that it is restrictive—the data structures are predetermined and cannot be optimized for any single implementation. Any small impact on the performance of a single implementation is, however, greatly outweighed by the advantages of this approach, especially in the case of a hybrid test generation program.

In closing, it is hard to imagine an environment more conducive to the development of a library of modular routines for ATPG than the one provided by the C programming language and the UNIX operating system. Because C is terse and deals with the same types of objects that computers do (e.g., characters, numbers and addresses), compilers can easily be developed which produce efficient machine language programs. The UNIX utilities enhance programmer productivity by providing the basic functions needed in a software development environment.

BIBLIOGRAPHY

- [1] T. Reid, *The Chip*, Simon and Schuster, New York, 1984.
- [2] G. Roth, *Fundamentals Of Logic Design*, West Publishing Company, St. Paul, Minnesota, 1979.
- [3] F. Tsui, *LSI/VLSI Testability Design*, McGraw-Hill, New York, 1987.
- [4] G. Moore, "VLSI—Some Fundamental Challenges," *IEEE Spectrum*, April 1979, pp. 30-37.
- [5] M. Breuer and A. Friedman, *Diagnosis & Reliable Design Of Digital Systems*, Computer Science Press, Rockville, Maryland, 1976.
- [6] A. Susskind, "Diagnostics For Logic Networks," *IEEE Spectrum*, October 1973, pp. 40-47.
- [7] A. Miczo, *Digital Logic Testing And Simulation*, John Wiley & Sons, New York, 1986.
- [8] J. Roth, "Diagnosis Of Automata Failures: A Calculus And A Method," *IBM Journal of Research and Development*, July 1966, pp 278-291.
- [9] P. Goel, "An Implicit Enumeration Algorithm To Generate Tests For Combinational Logic Circuits," *IEEE Transactions on Computers*, March 1981, pp. 215-222.
- [10] H. Fujiwara and T. Shimono, "On The Acceleration Of Test Generation Algorithms," *IEEE Transactions on Computers*, December 1983, pp. 1137-1144.

- [11] K. Mei, "Bridging And Stuck-At Faults," *IEEE Transactions on Computers*, July 1974, pp. 720-727.
- [12] T. Williams (Editor), *VLSI Testing*, North-Holland, Amsterdam, The Netherlands, 1986.
- [13] J. Hughes and E. McCluskey, "Multiple Stuck-At Fault Coverage Of Single Stuck-At Fault Test Sets," *Proceedings of the International Test Conference*, 1986, pp. 368-373.
- [14] J. Hayes, "Fault Modeling," *IEEE Design and Test of Computers*, April 1985, pp. 88-95.
- [15] M. Williams and J. Angell, "Enhancing Testability Of Large-Scale Integrated Circuits Via Test Points And Additional Logic," *IEEE Transactions on Computers*, January 1973, pp. 46-60.
- [16] E. Eichelberger and T. Williams, "A Logic Design Structure For LSI Testability," *Proceedings of the Design Automation Conference*, 1977, pp. 462-468.
- [17] T. Williams and K. Parker, "Design For Testability—A Survey," *Proceedings of the IEEE*, January 1983, pp. 98-112.
- [18] P. Winston, *Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, 1984.
- [19] J. Roth, W. Bouricius and P. Schneider, "Programmed Algorithms To Compute Tests To Detect And Distinguish Between Failures In Logic Circuits," *IEEE Transactions on Computers*, October 1967, pp. 567-580.
- [20] O. Ibarra and S. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Transactions on Computers*, 1975, pp. 242-249.

- [21] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- [22] R. Bennetts, *Design Of Testable Logic Circuits*, Addison-Wesley, Reading, Massachusetts, 1984.
- [23] Special Session, "Recent Algorithms For Gate-Level ATPG With Fault Simulation And Their Performance Assessment," *Proceedings of the International Symposium on Circuits and Systems*, June 1985.
- [24] F. Brglez, P. Pownall and R. Hum, "Accelerated ATPG And Fault Grading Via Testability Analysis," *Proceedings of the International Symposium on Circuits and Systems*, June 1985, pp. 695-698.
- [25] H. Min, personal communication, November 1988.

VITA

Thomas Humbert Belvin, Jr., was born in [REDACTED] on [REDACTED], the son of [REDACTED] and [REDACTED]. After graduating from Buchholz High School in Gainesville, Florida, in 1974, he entered Florida Technological University in Orlando, Florida. In 1976, he joined the United States Air Force, and returned to college at The University of Texas at Austin in 1978. After being awarded the degree of Bachelor of Science in Electrical Engineering from The University of Texas in December, 1980, he received a commission in the United States Air Force. During the following years he served as an electrical engineer in the Air Force, achieving the rank of captain. In August, 1987, he entered The Graduate School of The University of Texas.

Permanent address: [REDACTED]
[REDACTED]

This thesis was typeset¹ with L^AT_EX by the author.

¹L^AT_EX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's T_EX program for computer typesetting. T_EX is a trademark of the American Mathematical Society. The L^AT_EX macro package for The University of Texas at Austin thesis format was written by Khe-Sing The.